



University of
Pittsburgh

Algorithms and Data Structures 2

CS 1501



Spring 2023

Sherif Khattab

ksm73@pitt.edu

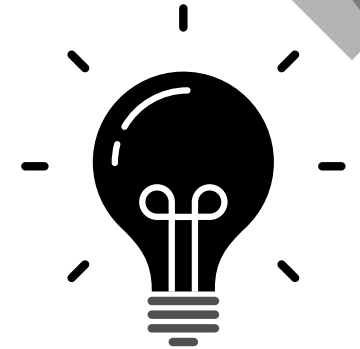
Announcements

- Homework 1 is due this Friday
- Lab 1 posted on Canvas
 - Explained in recitations of this week
 - Distributed using Github Classroom
 - Due on Tuesday 1/31
- Assignment 1 will be posted this Friday

Previous lectures ...

- A technique for modeling runtime of algorithms
 - $\sum_{all\ algorithm\ steps} Cost * frequency$
- Split the algorithm into blocks
 - steps in each block have the same frequency
 - $\sum_{all\ blocks} Cost * frequency$
- Asymptotic analysis
 - Focus on the order of growth of running time functions
 - Ignore lower order terms
 - Big O family of functions
 - and constant factors
 - Tilde Approximation
 - Keep the constant factor of the highest order term

Bonus Alert!



- Modify ThreeSum to work correctly with duplicate values in the input
 - The triples must be distinct in value
- Write a $\Theta(n^2)$ solution to the 3-sum problem

Send your solution using Piazza in a private message to all instructors labeled with the “bonus” tag

The Boggle Game

- Given a 4x4 board of letters, find all words with at least 3 adjacent letters
- Adjacent letters are horizontally, vertically, or diagonally neighboring
- Any cube in the board can only be used once per word
 - but can be used for multiple words



How to move through the Boggle letters?

- Have 16 different options to start from
- Have 8 different options from each cube
 - From $B[i][j]$:
 - $B[i-1][j-1]$
 - $B[i-1][j]$
 - $B[i-1][j+1]$
 - $B[i][j-1]$
 - $B[i][j+1]$
 - $B[i+1][j-1]$
 - $B[i+1][j]$
 - $B[i+1][j+1]$



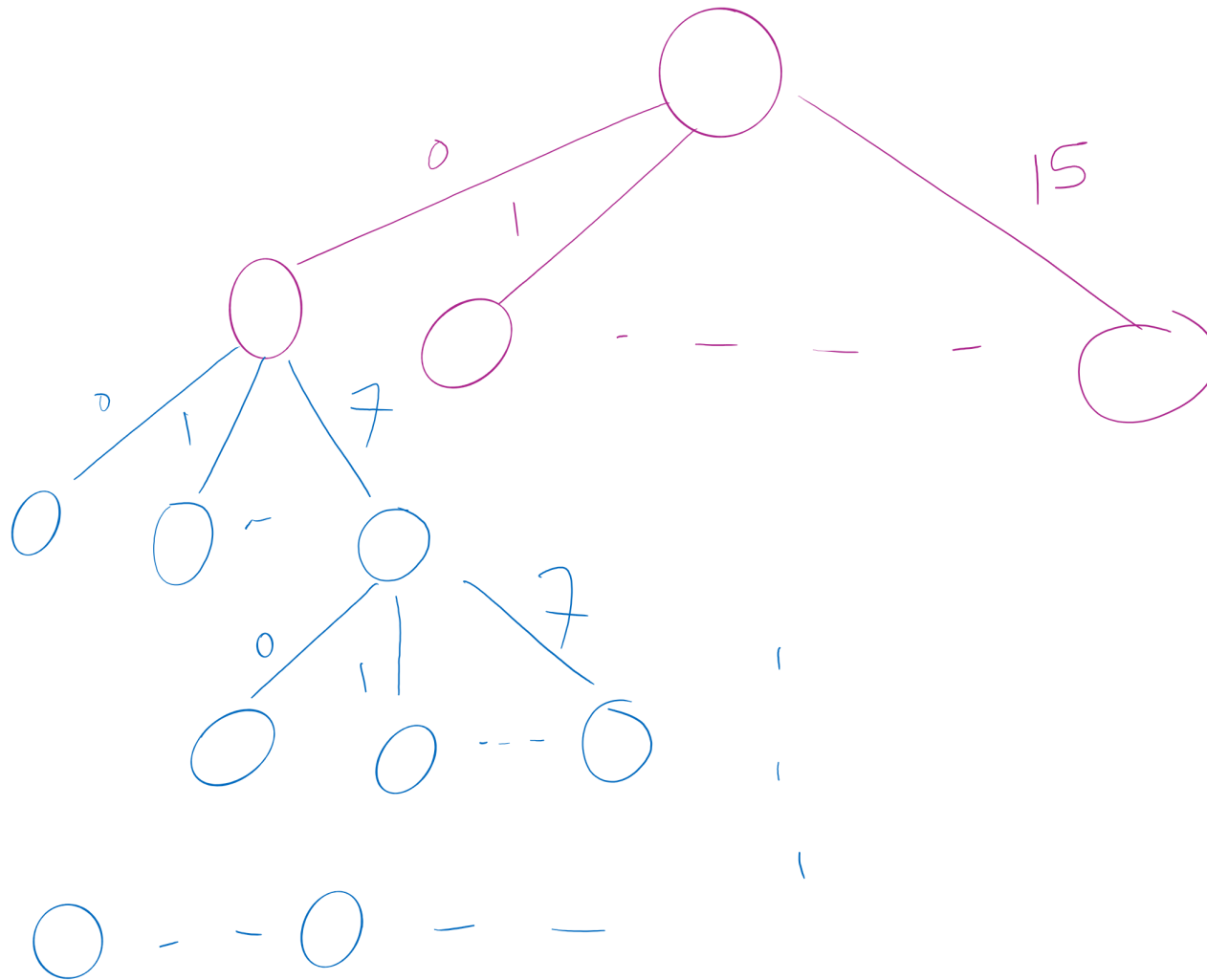
Boggle Game: Decisions and Options

- For the first decision (which cube to start from), how many possible options do we have?
 - 16
- Starting from a cube, the next decision to make is which adjacent cube to move to.
 - There are at most 8 possible options to choose from
- There is a decision to make at each cube that we go through
 - A maximum of 15 decisions
 - Some decisions may be trivial
 - the number of *valid* options < 2



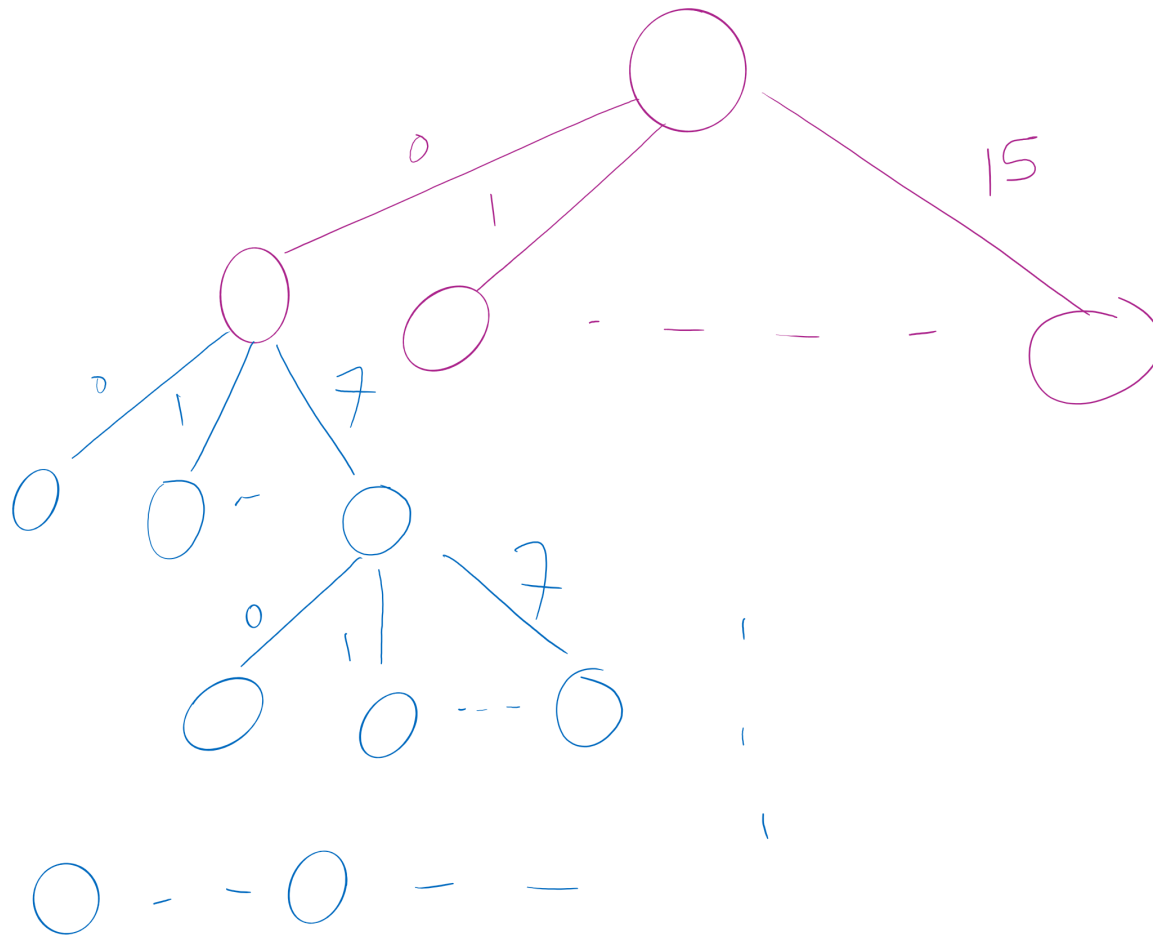
Search Space

Each node (circle) that has branches corresponds to a decision



Exhaustive Search Algorithm

- In an exhaustive search algorithm, we try all possible options for all possible decisions
 - Looking for one or all solutions



First decision

Are all 16 options *valid*?

Yes!

For next decisions, are all 8 options valid?

- No!
- Can't go past the edge of the board
- Can't reuse letter cubes
- Each letter added must lead to the prefix of an actual word
 - check the dictionary as we add each letter, if the currently constructed string does not start any word (i.e., not a prefix) don't go further down that way
 - Practically, this can be used for huge savings
 - This is called pruning!



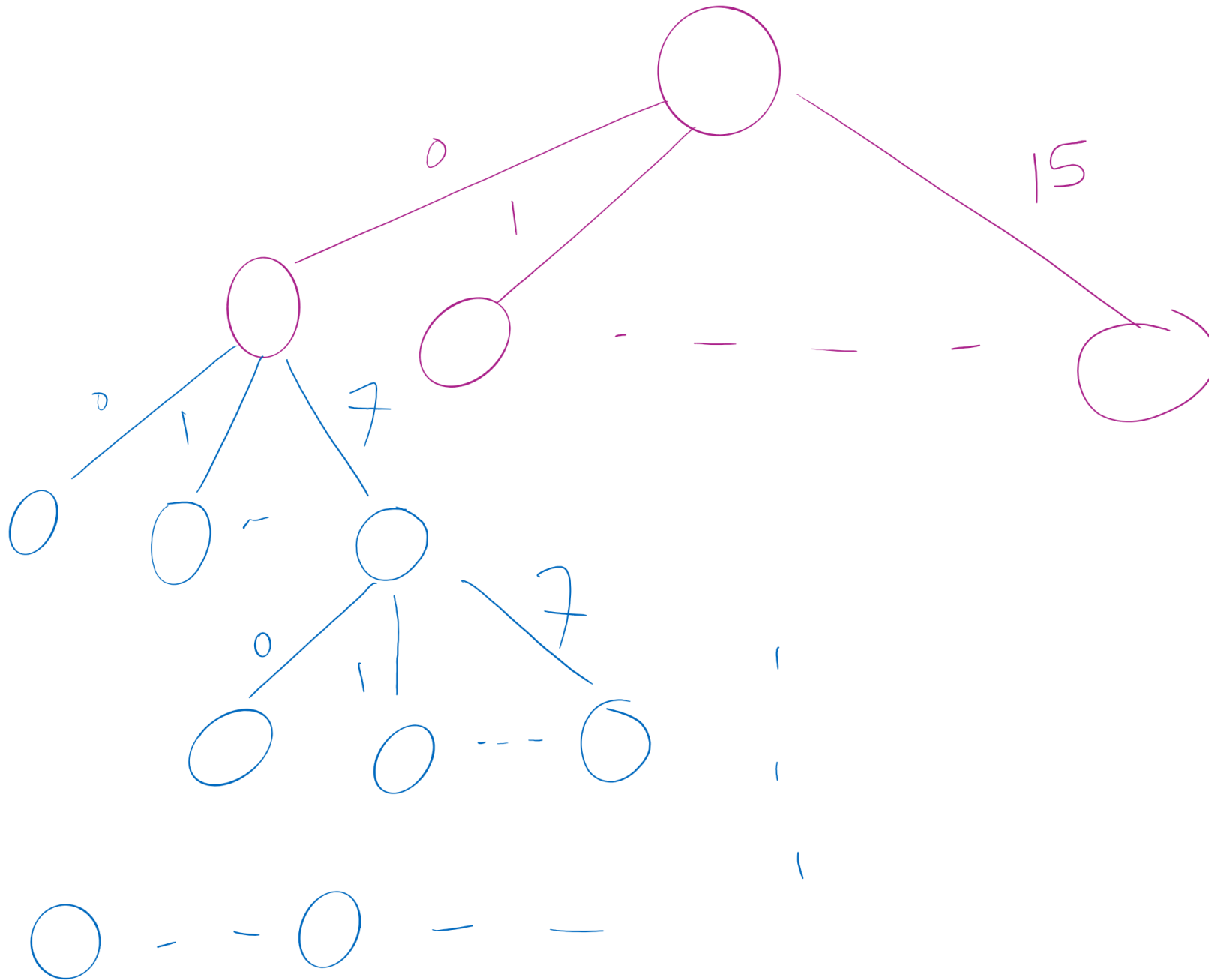
What happens when an options is valid?

- The same process pretty much repeats
- Check all 8 options and find a valid one to go down
- Do we need to go down just one valid option?
 - Or down all valid options?
- We need to somehow go back and try the other valid options
 - This is called backtracking

What happens when no valid options?

- We need to somehow go back to the previous decision and try other valid options if any
 - What if no more valid options for the previous decision?
 - Go back to the previous decision and so on ...
 - This is again called backtracking

Backtracking on the Search Space



Then what?

How can we code up such exhaustive search with backtracking and pruning?

Backtracking Framework

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            if partial solution a valid solution  
                report partial solution as a final solution  
            if more decisions possible  
                solve(next decision, updated partial solution)  
            undo changes to partial solution  
        }  
    }  
}
```


Application to Boggle Game

void solve(current decision,

Which cube are we at:

row number (0..3) and column number (0..3)

Backtracking Framework

```
void solve(....., partial solution) {  
    ....  
}
```

The letters that we have seen so far

Backtracking Framework

```
void solve(....) {  
    for each option at the current decision {  
        ...  
    }  
}
```

We have 8 options (except for the first decision, which has 16 options)

- From $B[i][j]$:
 - $B[i-1][j-1]$
 - $B[i-1][j]$
 - $B[i-1][j+1]$
 - $B[i][j-1]$
 - $B[i][j+1]$
 - $B[i+1][j-1]$
 - $B[i+1][j]$
 - $B[i+1][j+1]$

Backtracking Framework

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            ...  
        }  
    }  
}
```

Invalid options send us outside the board, reuse a cube, result in a non-prefix

Backtracking Framework

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            ...  
        }  
    }  
}
```

Append the letter to the partial solution and mark the cube as used

Backtracking Framework

...

if partial solution a valid solution

report partial solution as a final solution

...

If the partial solution is a 3+ letter word in the dictionary

Backtracking Framework

...

if more decisions possible

...

A decision is possible if partial solution is a prefix of a word in the dictionary

Backtracking Framework

...

`solve(next decision, updated partial solution)`

...

a recursive call.

**next decision is the next cube down the direction
that we chose**

Backtracking Framework

...

undo changes to partial solution

...

Remove the last letter that we appended from partial solution and mark cube as unused

Backtracking Framework

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            if partial solution a valid solution  
                report partial solution as a final solution  
            if more decisions possible  
                solve(next decision, updated partial solution)  
            undo changes to partial solution  
        }  
    }  
}
```

You will implement this algorithm in Lab 1 next week!

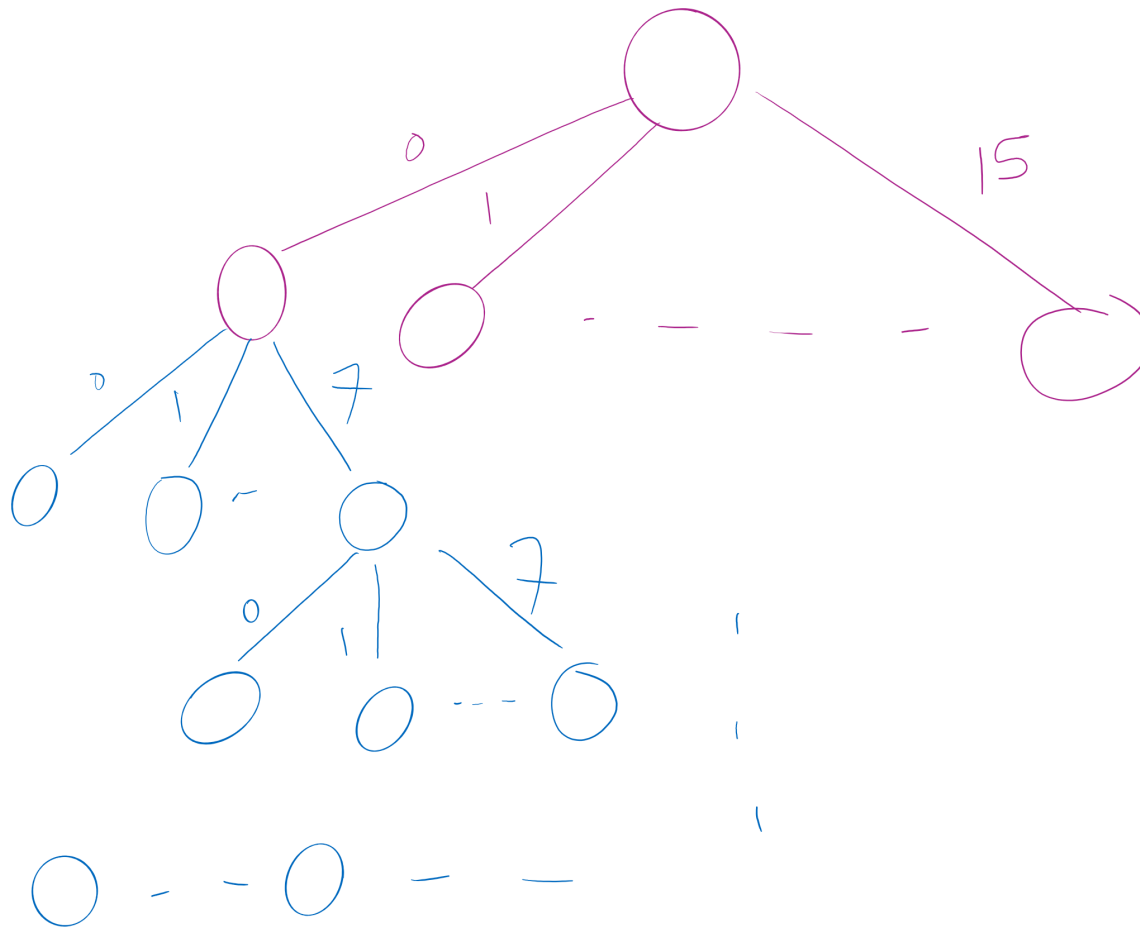
Backtracking Algorithm for Boggle

```
void solve(row and column of current cell, word string so far) {  
    for each of the eight directions {  
        if neighbor down the direction is a in the board and hasn't been used {  
            append neighbor's letter to word string and mark neighbor  
            as used  
            if word string a word with 3+ letters  
                add word string to set of solutions  
            if word string is a prefix  
                solve(row and column of neighbor, word string)  
            delete last letter of word string and mark neighbor as unused  
        }  
    }  
}
```

How many recursive calls are made by solve?

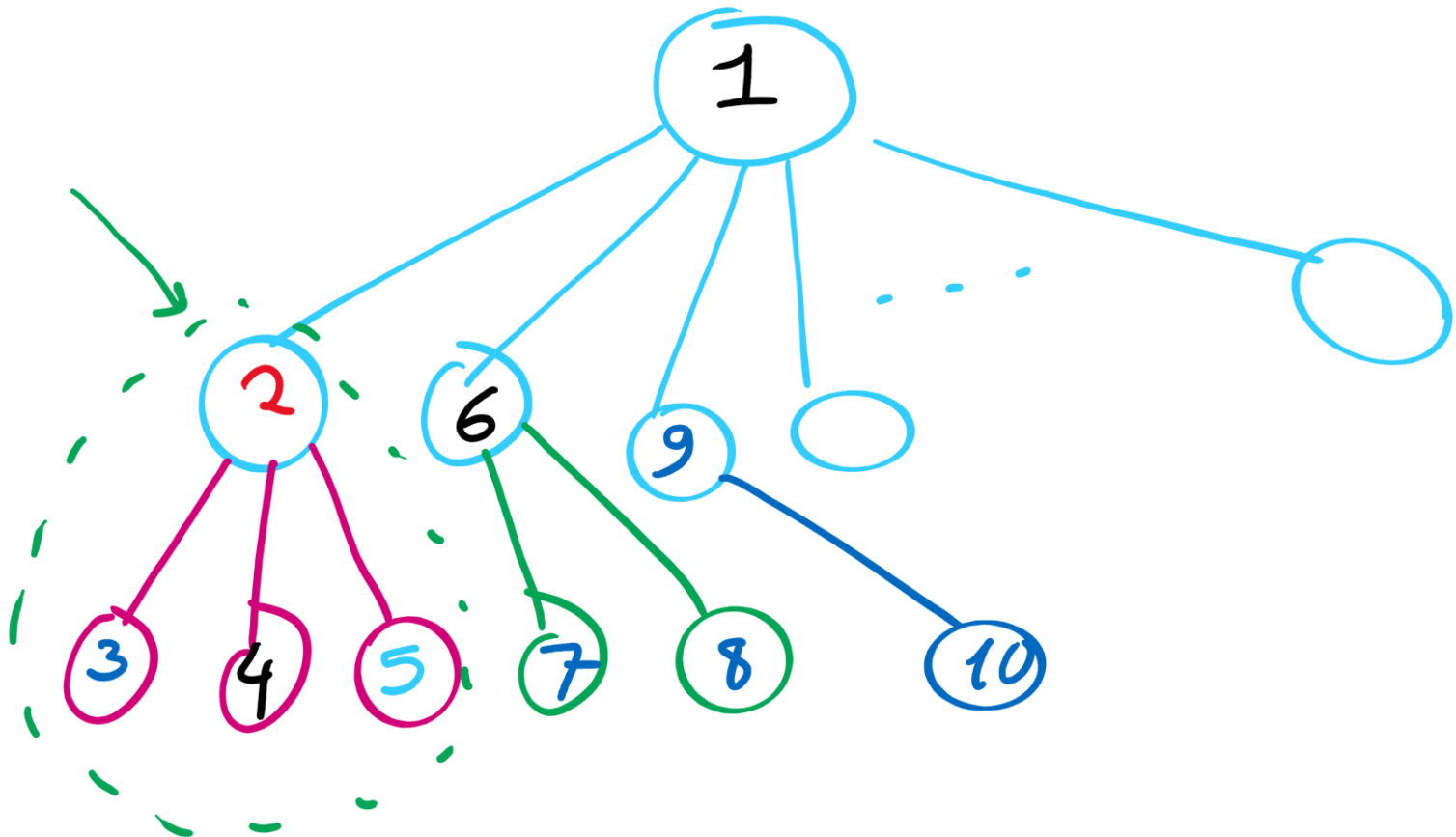
Search Space for Boggle

- The search space can be modeled as a *tree*
- Each circle (aka node) represents one call to solve
 - except the root node (why?)



Search Tree Traversal Order in Backtracking

- Each node (circle) corresponds to a recursive call
- The runtime cost per node is the cost of all statements except the recursive call



Moving down the tree

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            if partial solution a valid solution  
                report partial solution as a final solution  
            if more decisions possible  
                solve(next decision, updated partial solution)  
            undo changes to partial solution  
        }  
    }  
}
```

Backtracking (moving up the tree)

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            if partial solution a valid solution  
                report partial solution as a final solution  
            if more decisions possible  
                solve(next decision, updated partial solution)  
            undo changes to partial solution  
        }  
    }  
    return  
}
```

What is the running time?

- Can't easily use the frequency and cost technique because of the recursive call(s)
- In the worst case, the backtracking algorithm must visit each node in the search space tree
 - Why?
- At each node, the *non-recursive* part of solve executes
- The worst-case runtime =
 number of nodes *
 work per node (non-recursive part of solve)

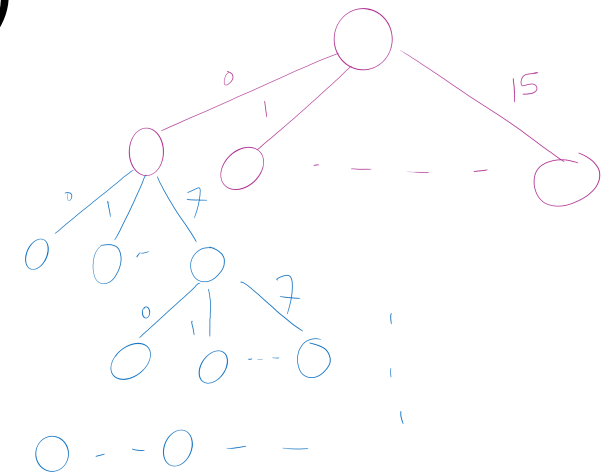
Non-recursive part of solve

Everything but the recursive calls

```
void solve(current decision, partial solution) {  
    for each option at the current decision {  
        if option is valid {  
            apply option to partial solution  
            if partial solution a valid solution  
                report partial solution as a final solution  
            if more decisions possible  
                solve(next decision, updated partial solution)  
            undo changes to partial solution  
        }  
    }  
}
```

Search Space Size

- How many nodes are there?
- **Maximum number of nodes** = $1 + 16 * (??)$
- $= 1 + 16 * (1 + 8 + 8^2 + 8^3 + \dots + 8^{15})$
- $= 1 + 16 * \theta(\text{largest term})$
- $= 1 + 16 * \theta(8^{15})$
- In terms of board size (n)
 - **Maximum number of nodes** = $1 + n * \theta(8^{n-1})$
 - $= \theta(n * 8^{n-1}) = \theta(n * 8^n)$
- Exponential!



Running time

- Worst-case runtime of Backtracking for Boggle = $\Omega(n \cdot 8^n)$, where n is the board size (# cells)
 - if the non-recursive work is constant, that is $O(1)$, then
 - worst-case runtime = $O(n \cdot 8^n)$
 - We will see we make it constant
- Worst-case runtime of backtracking algorithm is exponential!
 - Pruning has practical savings in runtime, but doesn't significantly reduce the runtime
 - Still exponential in the worst case

Non-recursive Work

```
void solve(row and column of current cell, word string so far) {  
    for each of the eight directions {  
        if neighbor down the direction is a in the board and hasn't been used {  
            append neighbor's letter to word string and mark neighbor used  
            if word string a word with 3+ letters  
                add word string to set of solutions  
            if word string is a prefix  
                solve(row and column of neighbor, word string)  
            delete last letter of word string and mark neighbor as unused  
        }  
    }  
}
```